Scheduling Task and Data Parallelism in Array Languages with Work Assisting

Ivo Gabe de Wolff ✉[0000−0002−4731−2234],
David van Balen[0000−0002−2807−9860], and
Gabriele K. Keller[0000−0003−1442−5387]

Utrecht University

Abstract. High level languages for parallelism need to be performant on a wide range of workloads: they may be data-parallel and/or task-parallel, as well as regular or irregular. Scheduling, which is implemented via an interaction between the runtime system and the generated code, has a significant impact on the performance and scalability of these languages. In this paper, we demonstrate the integration of Work Assisting, our dynamic scheduler combining task-parallel and data-parallel schedulers, in combinator-based parallel array languages. These languages require fusion for high performance, and often feature scans to support irregular computations. Chained scans, the fastest parallel scans in our experiments, require a data-parallel scheduler as provided by Work Assisting. We show how code can be generated with support for fusion and chained scans, which can also fuse better than classic three-phase scans. We present the integration of Work Assisting into an actual compiler and runtime system of a such a language, Accelerate, and evaluate its performance in this context for a range of applications.

Keywords: Functional array languages · Scheduling · Code generation.

1 Introduction

Functional parallel array languages, such as APL [16], DaCe [31], Futhark [15], SaC [14], Lift [26] and Accelerate [20], aim at providing the programmer with a convenient programming model, which provides portability over a range of architectures, while still achieving performance which is as close to hand-optimized, architecture specific implementations as possible [28]. However, in contrast to customized, highly specialized solutions, the compilers and runtimes of these high-level languages cannot assume much about a program. They therefore have to employ strategies which work well on a large range of programs, which is especially the case for scheduling: the system should distribute the parallel workload efficiently, whether the program contains task and/or data parallelism, as well as whether it is regular or irregular. In parallel languages, scheduling is implemented via an interaction between the runtime system and the generated code.

In the context of code generation, compilers for array based functional languages have to solve two concrete problems: they have to fuse array traversals,

often expressed in the high-level language via higher-order operations, such as maps, scan, folds and permutations, to generate the optimal number of loops. Furthermore, efficient code has to be generated for the different communication patterns, even when they are fused with other patterns. Among these, scans, also known as generalised prefix sums, are the most challenging. As scans are used extensively when expressing irregular parallel algorithms in array languages, an efficient parallel implementation for scans is particularily important.

Given the importance of scheduling, it is not surprising that a wide range of strategies have been proposed, and the choice of the strategy influences which parallel algorithms can be expressed in the language. Work stealing [7] or variants of it are commonly used in parallel languages and libraries, for instance in OneTBB [24], Halide [23] and Manticore [13]. It is a task-parallel scheduling strategy, but it can also be used for data parallel applications by splitting a data-parallel computation in multiple tasks [27]. However, some data-parallel algorithms require a dedicated data-parallel scheduler; for instance, chained scans [12, 21, 11] require that blocks of an array are distributed or claimed in a specific order. Another reason why explicit data-parallel schedulers might be chosen is that they potentially have smaller overhead than task-parallel schedulers for purely data-parallel computations. For this reason, some languages and libraries, including SaC [14] and early versions of OpenMP [1], have thus chosen to use a data-parallel scheduler, and those cannot exploit task parallelism. Work Assisting [9] is a scheduler for mixed data and task parallelism: it uses a task-parallel scheduler for task parallelism, and a data-parallel scheduler within a data-parallel subcomputation.

In earlier work [9], we have introduced the Work Assisting scheduler and evaluated its performance in stand-alone benchmarks. Since then, we implemented Work Assisting in the compiler of Accelerate to futher investigate the integration of Work Assisting with code generation, fusion, and other optimizations in a compiler, and to perform benchmarks on larger programs. We report on these findings in this paper; we discuss how suitable Work Assisting is as scheduling strategy for a parallel array language, and how it interacts with code generation with fusion. We compare the performance of Accelerate with Work Assisting to that of Futhark [15], a performant stand-alone parallel language with a very similar programming model, as well as hand-optimized, base-line implementations. The implementation we present in this paper is able to:

- handle task and data parallelism,
- support parallel chained scans [12, 21, 11],
- support aggresive fusion [29], and
- perform no dynamic allocations for scheduling (except for the task queue).

We discuss how these together help to achieve the desired high performance for data parallel array languages.

The remainder of this paper is structured as follows. After laying a background on parallel array languages in Section 2, we introduce the Work Assisting runtime in Section 3. We elaborate on code generation for the data-parallel sub-

computations, which we call *kernels* following GPU terminology, in Section 4. We present benchmarks in Section 5.

2 Preliminaries

There are several kinds of parallelism in functional array programming. Each of these require special handling and are supported to various degrees in the different languages. First, we have the array operations which operate in parallel over the values of the array. We call the parallelism within an array operation *data parallelism*. These operations may be simple maps of sequential operations over one-dimensional or multi-dimensional arrays, or operations that require some additional communication like folds, scans or permutations. Compilers typically require that array operations are not started from within other array operations, either by language design or a compiler pass [4].

Due to the absence of side-effects, purely functional languages in general expose another potential source of parallelism: computations which have no data dependency can be evaluated in parallel, without affecting the result. We call the parallelism between array operations *task parallelism*. This implicit task parallelism can be exploited by the compiler to reduce the overhead of data-parallel operations like scans that do not scale linearly, or keep processors busy when the explicit data parallelism available is not sufficient. Since Accelerate is embedded in Haskell, task parallelism in Accelerate may also occur when a Haskell program runs multiple Accelerate programs.

Since only the outer level of the program may contain task parallelism, acting as a wrapper around data-parallel computations, we can compile the task-parallel and data-parallel parts of the program separately. We will call the compiled data-parallel computations *kernels*, as our model is similar to the GPU model.

2.1 Fusion

Fusion, a program transformation which combines several traversals of a data structure, is an important optimization to reduce the memory footprint and execution time of a program. This is especially important for functional array languages, as they advocate constructing a program using small building blocks like parallel maps, folds and scans. Unfortunately, in the parallel context, the problem is more difficult, since the parallel structure needs to be maintained. We can split fusion into two subproblems: deciding what operations should be fused, and generating code for a fused set of operations. Accelerate uses an Integer Linear Programming formulation for the former [29]. In this paper we only focus on the latter and present how we generate code for a list of operations that are fused together. There are two relevant consequences of Accelerate's fusion system for this paper. Firstly, operations may be *vertically fused*, in which case the corresponding array is not manifest in memory. Instead, a produced value is stored in a scalar variable and is directly consumed by another array operation. Secondly, it is not practical to design specialized skeletons or templates for some

of the combinators, as Accelerate previously did [8]. Instead, we need a single template that we can instantiate to express any combination of array operations.

2.2 Chained Scans

Scans, generalized prefix sums, are an important parallel primitive for many, especially irregular, applications. At each index, they compute the combined value of all prior values. We found that chained scans [12, 21], whereas originally designed for GPUs, are also the fastest scans on multi-core CPUs [11]. These scans operate with a single traversal over the data, in contrast to the more commonly used reduce-then-scan. We explain the working of chained scans together with the code we generate for them in Section 4.2.

Scans are commonly used for compaction or filtering, which can be implemented using a scan followed by a scatter (permute), as shown in Listing 1. Using chained scans and a sufficiently flexible fusion system, a filter can be implemented using a single traversal over the input [21]. To enable this, we must allocate an array with the size of the input for the output, and shrink it to the correct size later, as we do not know the output size in advance.

3 Work Assisting Runtime

The core idea behind our Work Assisting scheduler [9] is to primarily use the task-parallel scheduler. Only when the system runs out of tasks, threads try to *assist* in a kernel (a data-parallel computation) of another thread. These kernels can be found via a shared array called *activities*, where threads share the kernel they are currently working on. This array has one slot per thread. Scheduling within a kernel happens with a dynamic data-parallel scheduler. In the current implementation of Accelerate, we use self scheduling [17] with atomic fetch-and-add: when a thread needs to claim new work within a data-parallel kernel, it does so by atomically incrementing a shared counter for that kernel.

Most important types and definitions of the runtime can be found in Listing 2.

We introduced Work Assisting in [9]. However, the Accelerate implementation which we describe and evaluate here differs in a few minor ways: In the original description, a task can be a data-parallel task, while a task in the Accelerate

Listing 1 Simplified definition of filter, which can be fused into a single kernel except for the shrink at the end.

```
function filter(predicate, input)
  ps ← map(\x → if predicate x then 1 else 0, input)
  (destination, output_size) ← scanl'((+), 0, ps)
  ws ← zipWith3(\x p d → if p then Just (d, x) else Nothing, input, ps, destination)
  output ← Uninitialized array with the same size as input
  scatter(output, ws)
  return shrink(output, output_size)
```

implementation *returns* a data-parallel computation, which we call a *kernel*. We handle *EmptySignal* outside of the work function in the runtime, to simplify code generation. Furthermore, we do not store *work_size* explicitly; instead it is computed in the work function and only available within that function. The finish function is not explicitly stored; it is now handled by calling the work function with a magic value and by storing a continuation task in the kernel. We changed *work_index* and *work_size* to 64-bit integers.

3.1 Program as a Coroutine

Array languages have a clear separation between the data-parallel parts of the program (kernels) and the control flow between them. The outer level of a program consists of the control flow and administrative work, and may contain task parallelism. To support task parallelism in this outer level, we compile this into a coroutine [22], a function that can be suspended during its execution. We suspend the coroutine when we launch a kernel and the execution of the coroutine is resumed after the kernel is finished. It is also suspended when we need to synchronize with another part of the program, e.g. when we need to join the execution after a fork.

Our implementation is similar to the stackless coroutines in for instance Rust [19]. The variables of the coroutine are stored in an object on the heap which we call a Program (Listing 2). This object also contains a pointer to the function of the program (the *program function*) and a reference count.

When suspending the function, the system marks where (at which state or location in the program) the function should resume. Whereas this is typically stored in the object, Program in our case, we store it outside of that object, in a task. A task is then a pointer to a program and a location in that program. This allows the coroutine to be in multiple states concurrently. This allows us to implement *fork* without allocations: a fork is implemented by scheduling a task of the same program, at a different location. The coroutine is compiled into the program function and takes the Program and the location in the program as arguments.

Listing 2 Core Types for the Runtime System

```
struct Workers
    int thread_count;
    TaskScheduler* task_scheduler;
    KernelLaunch* activities;
struct Program
    int reference_count;
    ProgramFunction run;
    data;    // Variables of the program
type ProgramFunction
    (Workers*, int thread_index, Program*, int location) → KernelLaunch*
struct Task
    Program* program;
    int location;
function schedule(Workers*, Task)
function schedule_after(Workers*,
    Task, Signal, SignalWaiter)
function signal_resolve(Workers*,
    Signal)
```

Synchronization within a program is implemented using a Signal, which is similar to a future or promise in other languages. Tasks may wait on a signal, until another task resolves the signal. A signal can be resolved once and does not contain an actual value. Waiting tasks are stored in a concurrent linked list of SignalWaiter objects. These objects are preallocated in the program for each point in the program where it waits on a signal.

3.2 Kernels

When a program needs to execute a data-parallel computation, it returns an object describing the launch of that kernel. The program object contains a pre-allocated kernel launch object for each kernel in the program, and we can thus start a kernel without memory allocations.

The kernel launch object contains a fixed form header, and the input and output arrays of this kernel and their sizes. If the operations in the kernel require it, we can also allocate additional memory called *kernel memory* in that object, for instance for communication between threads for a scan.

In the header, we store a *kernel function*, the function that all threads working on this kernel should call. As its arguments, this function gets a pointer to the kernel launch object and an index *first_ index*, denoting the index of the tile we should first work on. The header also contains the task which could be executed after the kernel.

The work within a kernel is split in tiles. Argument *first_ index* is the index of the tile that this thread should work on first. The Work Assisting runtime has already claimed this tile. Later tiles are claimed by the work function itself via atomic fetch-and-add. The function is also called with two magic values. It is called with $\mathit{first\_\ index} = -1$ before the parallel work of this kernel starts, where it may initialize kernel memory. Finally, it is called with $\mathit{first\_\ index} = -2$ after the parallel work of the kernel, for instance to write the result of a fold to the output array when all threads have finished.

4 Code Generation for Data Parallelism

Now that we have seen the runtime system and the method of generating code for the task-parallel part of the program, we can focus on the data-parallel kernels in the program. The work of a kernel is scheduled via tiles: the workload is split into tiles, which are claimed one by one via atomic fetch-and-add. Whereas some data-parallel combinators are more naturally implemented by splitting the load in a fixed number of tiles, and some in fixed size tiles, we always use fixed size tiles for uniformity. This way we can compile all the combinators into one generic form. This is needed for fusion, as any combinator can be fused with any other combinator.

Fixed size tiles are required for chained scans [12], as they expect that the data of a tile fits in the cache. The structure of our generated code is mainly dictated by our support for chained scans. We first present this general form, and then explain how the array combinators can be compiled into this form.

4.1 Generic Format for Code Generation

We present the generic structure into which we can compile any array operation in Listing 3. The highlighted lines are the places where operation specific code can be inserted. We first explain this generic structure, and then show how how concrete operations emit code at what parts of this structure in Section 4.2.

A kernel is compiled into a function and an accompanying object that contains the data (references to the input and output of the kernel and their sizes) and administrative information for the kernel. Array combinators may also reserve space in this object (label **A**). We call this *kernel memory*, which may be used for communication between the threads working on this kernel.

Listing 3 Template for a kernel, with tile size 2048.

When compiling an array combinator, code may be placed at the highlighted positions.

```
struct Arg
    KernelFunction* work_function; Program* program; int program_location;
    int active_threads; int work_index;
    A: Declare kernel memory
function kernel(Arg* arg, int first_index)
    tile_count ← (arg→n + 2047)/2048
    if first_index = −1 then
        B: Initialize kernel memory
        return tile_count > 1
    else if first_index = −2 then
        C: Finalize kernel
        return
    previous_idx ← −1; sequential ← true; tile_idx ← first_index
    D: Thread initialization
    while tile_idx < tile_count do
        if tile_idx = previous_idx + 1 then sequential ← false
        start ← tile_idx ∗ 2048; end ← min(start + 2048, arg→n)
        if sequential then
            E': Before tile loop
            for i in start . . . end do   // Tile loop
                F': Body of tile loop

            G': After tile loop
        else
            E: Before tile loop
            for i in start . . . end do   // Tile loop
                F: Body of tile loop

            G: After tile loop
            for i in start . . . end do   // Tile loop
                H: Body of next tile loop
        tile_idx ← atomic_fetch_add(arg→work_index)
    I: Thread finalization
```

Before and after the parallel work of a kernel, the kernel function is called to initialize the launch (with *first_ index* $= -1$) and finalize the kernel (*first_ index* $= -2$). Array combinators can emit initialization and finalization code respectively at label **B** and **C**. A fold for instance uses this to set the reduced value to zero at **B** and write the final result to the output at **C**. After initialization, a thread returns whether the kernel may be executed in parallel. If the input is small, the kernel function will decide to not run the work in parallel.

To perform the parallel work, multiple threads will call the work function. Array combinators may place code to initialize a thread at label **D**. Afterwards a thread will repeatedly handle a tile. It first works on tile *first_ index*, claimed by the Work Assisting runtime, and later claims tiles using atomic fetch-and-add. Within that while loop, we traverse the data in that tile in a *tile loop*. We can place code before (**E**), within (**F**) and after the tile loop (**G**). We allow combinators to insert specialized code for sequential execution (**E'**, **F'** and **G'**). As long as a thread is the only thread to work on a kernel, it will operate in a sequential mode, and as soon as another thread also works on this kernel, it will switch to the parallel mode. Only scans generate different code, as they require two tile loops in the parallel mode. We thus allow combinators to also generate code in a later tile loop (**H**). Finally, combinators may emit code to finalize a thread (**I**). For a commutative fold, we use this to let a thread contribute its locally reduced value to the global value.

All operations in a kernel are compiled to this format, and are placed together in one kernel. If an operation declares that it needs a second tile loop (by generating code at label **H**), later operations will also be placed in that tile loop. If a kernel contains multiple of those operations, it will thus have more than two tile loops. If a kernel only has a single tile loop in the parallel mode, the single-threaded mode is discarded, as it would already not have single-threaded overhead.

4.2 Generating Code in the Generic Format

To illustrate that the array combinators in Accelerate can be compiled to this format, we demonstrate this translation for several combinators. First, embarrassingly parallel combinators like map and generate only need to generate code at labels **F** and **F'**.

Scans Scans are the most complicated combinators. We show the generated code corresponding with $(ys, z) =$ scanl' $(+)$ 0 xs in Listing 4, which is an exclusive left-to-right scan where the prefix values of xs are stored in ys, and the total reduced value in z. Scans require synchronization between the threads, for which they need two variables in kernel memory, defined at label **A** and initialized at label **B**. Variable *scan_ index* denotes the index of the tile that should next be incorporated in the prefix value in *scan_ prefix*.

If we ignore the special sequential mode for sequential execution, the scan requires two tile loops to handle a tile of the input. In the first tile loop, it reduces the values in the tile to an aggregate (**E** and **F**). It then waits for

its turn to update the *scan_prefix* (**G**). With the found prefix value, it then performs a scan over this tile in another tile loop (**H**). This chained scan thus needs to block between the two tile loops, but in return it gets better memory performance as the values of *xs* of this tile are likely to still be in cache in the second tile loop. The chained scan may be changed to a chained scan *with decoupled look-back* [21], to reduce the impact of the synchronization at label **G**. We are also considering to implement that in Accelerate.

To assure that the parallel chained scan executed on a single core performs as fast a sequential scan, we add a special sequential mode [11]. This is important when the program also contains task parallelism, as it then might be sufficient to only exploit task parallelism. Variable *sequential* tracks whether this was the only thread working on the scan. The sequential mode operates with one tile loop, where it can directly execute the scan (labels **E'**, **F'** and **G'**).

Fold Folds, or reductions, can be compiled in two ways. Since it is common that the operator of a fold is commutative, we have a specialized implementation of folds for commutative folds. Note that it is always required that the operator is associative for parallel execution, but commutativity is not required.

If the operator of a fold is not (known to be) commutative, we generate code via the same pattern as the code of parallel scans. This *chained fold* has more overhead than two-step folds due to the additional synchronizations, but it does allow us to generate code in the generic format. When we implement scans with a decoupled look-back [21], we also expect to lower the overhead of folds as well.

Listing 4 Code Generation for Exclusive Scan, $(ys, z) = \text{scanl'}\ (\oplus)\ 0\ xs$

```
A  int scan_prefix; int scan_index;
B  arg→scan_prefix ← 0
   arg→scan_index ← 0
C  *(arg→z) ← arg→scan_prefix
E' prefix ← arg→scan_prefix
F' arg→ys[i] ← prefix
   prefix ← prefix ⊕ arg→xs[i]
G' arg→scan_prefix ← prefix
   arg→index ← tile_idx + 1
E  aggregate ← 0
F  aggregate ← aggregate ⊕ arg→xs[i]
G  while arg→scan_index ≠ tile_idx do
     // Wait
   prefix ← arg→scan_prefix
   arg→scan_prefix ← prefix ⊕ aggregate
   arg→scan_index ← tile_idx + 1
H  arg→ys[i] ← prefix
   prefix ← prefix ⊕ arg→xs[i]
```

Listing 5 Code Generation for Commutative Fold, $y = \text{fold}\ (\oplus)\ 0\ xs$

```
A  int fold_lock;
   int fold_accumulator;
B  arg→fold_lock ← 0
   arg→fold_accumulator ← 0
I  while atomic_swap(&arg→fold_lock, 1) ≠ 0 do
     // Wait
   arg→fold_accumulator ← arg→fold_accumulator ⊕ local
   arg→fold_lock ← 0
C  *(arg→y) ← arg→fold_accumulator
D  local ← 0
F  local ← local ⊕ arg→xs[i]
F' local ← local ⊕ arg→xs[i]
```

The generated code of a commutative fold is shown in Listing 5. Since the operator is commutative, we may compute a local result per thread and at the end of the work for this thread, incorporate that local result in the global result (**I**). This happens once per thread instead of once per tile. The global value is stored in kernel memory, together with a lock (**A**).

4.3 Fused Away Arrays

By fusion, some arrays do not have to be fully manifest in memory. Instead, we introduce local variables for these arrays that either contain a single value or one value per element of a tile, depending on whether the array is used in multiple tile loops. We introduce this at **D** using *alloca*, to allocate them on the stack. LLVM may change this stack allocation to a variable stored in a register.

5 Benchmarks

To investigate the performance of Accelerate with the new Work Assisting scheduler and support for chained scans, we performed benchmarks on various applications taken from CFAL, Comparing Functional Parallel Array Languages [28]. We compare the performance of Accelerate with a reference implementation not in an array language, and Futhark, a stand-alone functional parallel array language. We ran the benchmarks on a machine with an AMD 2950X processor (16 cores, 32 threads) with 32GB of RAM on Ubuntu 22.04.

5.1 N-body and MultiGrid

We first briefly discuss two benchmarks of CFAL, before we discuss a detailed case study on the Quickhull benchmark. *N-body* is a naive implementation of n-body simulation and *MultiGrid* (MG) computes a solution to a differential equation. The latter was taken from the NAS benchmarks [3]. The implementations are available online[1] and we present the results in Table 1.

The n-body benchmarks show that the Work Assisting scheduler scales well in Accelerate. All languages scale well in the medium and large input, but Futhark and the OpenMP implementation scale worse on the small input. Scheduling is more difficult here, as there is less data parallelism available. Accelerate scales better than the other implementations here.

The OpenMP reference implementation of MG in Fortran is highly specialized, and these specializations are not directly possible in high level array languages. Their compilers cannot apply the same optimizations automatically, which gives them a large disadvantage in this application. As for the topic of this paper, both Futhark and Accelerate scale well to 32 threads. However on class C, Accelerate performs worse, both single-threaded and on 32 threads.

[1] https://github.com/ivogabe/CFAL-bench-new-accelerate or [10]

N-body	**Small**		**Medium**		**Large**	
Input size	10^3		10^4		10^5	
Steps	10^5		10^3		10^1	
Threads	1	32	1	32	1	32
OpenMP	190 s	23.5 s	232 s	13.7 s	193 s	13.6 s
Futhark	157 s	26.0 s	158 s	12.1 s	167 s	12.0 s
Accelerate	162 s	14.6 s	162 s	11.6 s	162 s	11.4 s
MG	**Class A**		**Class B**		**Class C**	
Input size	256×256×256		256×256×256		512×512×512	
Steps	4		20		20	
Threads	1	32	1	32	1	32
OpenMP	0.66 s	0.46 s	2.92 s	2.26 s	21.3 s	17.4 s
Futhark	3.65 s	0.68 s	15.5 s	3.00 s	127 s	24.3 s
Accelerate	3.68 s	0.67 s	17.0 s	3.11 s	168 s	51.2 s

Table 1. N-body and MultiGrid benchmarks

5.2 Quickhull

Quickhull is an algorithm to compute the hull of a set of points, in our case two dimensional points. Its structure is similar to quicksort, as it recurses twice on smaller sets. However, depending on the shape of the input, these two sets might together be significantly smaller than the input and the recursion depth thus depends on the shape of the input. Hence we measure the performance on three different input shapes: rectangle, circle and quadratic. For these inputs, the points were sampled from a rectangle or circle, or near a quadratic curve.

This application has nested parallelism, as the two parallel recursive calls may perform more parallel work. There are various ways to implement this in a parallel array language. Flattening [6] converts nested parallelism into flat parallelism.To further investigate the performance of Accelerate with its new scheduler and chained scans, we have implemented an entirely flattened implementation (**Flat**), and various implementations that do use task parallelism and are partially flattened. **Split** is flattened after an initial split, and implementations **Rec 2** and **Rec 5** use task-parallel recursion for the first two or five levels, and are flattened afterwards. All implementations make heavy use of scans.

We report the execution times of these in Table 2, together with two reference implementation: one in ParlayLib [5], that uses task parallelism instead of flattening, and the implementation in Futhark from the CFAL project, which is entirely flattened. The code is available online[2]. Our implementations are faster than Futhark, both single-threaded and on 32 threads. Futhermore, they scale more than Futhark. They do not scale as well as the reference implementation in ParlayLib. However, that implementation is not in an array language and manually reuses memory. Due to the functional nature of Accelerate, we do not want to concern the user with memory management. Instead, we should let the compiler optimize the program with for instance in-place updates.

[2] https://github.com/ivogabe/quickhull-benchmarks or [10]

	Rectangle		Circle		Quadratic	
Input size	25 M		25 M		25 M	
Output size	48		1057		358 K	
Recursion depth	7		11		21	
Threads	1	32	1	32	1	32
Reference implementations						
ParlayLib	2.18 s	0.096 s	2.13 s	0.14 s	12.2 s	1.58 s
Futhark	1.89 s	0.56 s	1.64 s	0.54 s	10.9 s	3.38 s
Accelerate						
Flat	1.06 s	0.26 s	1.00 s	0.30 s	8.36 s	2.29 s
Split	0.80 s	0.25 s	0.96 s	0.29 s	8.05 s	2.29 s
Rec 2	0.75 s	0.19 s	1.00 s	0.23 s	8.23 s	2.17 s
Rec 5	0.79 s	0.19 s	0.94 s	0.22 s	8.03 s	1.96 s

Table 2. Benchmarks of various implementations of Quickhull. Lower is better.

6 Related Work

Runtimes for parallel languages and frameworks are often build around work stealing [7], as we discussed in the introduction. Work stealing stores task in a queue per thread. This reduces contention on a queue, and improves cache coherency by primarily letting threads take tasks from their own queue. Only if that queue is empty, it will try to *steal* tasks from other threads. Work stealing may be used as the task-parallel scheduler in Work Assisting.

Other schedulers for mixed data and task parallelism do exist [30, 25], but they are less flexible: these schedulers fix the number of threads for a kernel at its start, in contrast to our scheduler.

To implement data parallelism via task parallelism, the data parallel workload needs to be split in tasks. The granularity of the parallel work influences the scheduling overhead [18, 24]. When implementing data parallelism via task parallelism, tasks should not be split into too many tasks, as that gives scheduling overhead, whereas too few tasks may result in work imbalance between the cores, especially for irregular computations. Lazy Binary Splitting [27] only splits work further depending on whether the local task queue is empty, and Heartbeat scheduling [2] only shares new tasks at fixed intervals, to prevent scheduling too many tasks. For data parallel schedulers like self scheduling [17], granularity must also be controlled. Smaller tile sizes might however be possible since claiming a tile requires only a single atomic increment and no allocations.

7 Conclusion

In this work we show the suitability of Work Assisting for parallel array languages. By employing both a task- and a data-parallel scheduler, we can efficiently exploit both forms of parallelism in array programs. We elaborated on the integration between the runtime and the generated code, via coroutines for

the task-parallel part of the program, and the template for code generation of data-parallel kernels. We support parallel chained scans, and can fuse them more than other fusion systems. Using our implementation of Work Assisting in Accelerate, we evaluated the performance in various benchmarks. Our implementation is often competitive with or faster than baseline implementations.

Artifact Availability The artifact is available in the Zenodo repository [10].

Disclosure of Interests. We have no relevant competing interests to declare.

References

1. OpenMP Application Programming Interface, version 5.2. Tech. rep. (Nov 2021), https://www.openmp.org/specifications/
2. Acar, U.A., Charguéraud, A., Guatto, A., Rainey, M., Sieczkowski, F.: Heartbeat scheduling: Provable efficiency for nested parallelism. In: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 769–782 (2018)
3. Bailey, D., Harris, T., Saphir, W., Van Der Wijngaart, R., Woo, A., Yarrow, M.: The nas parallel benchmarks 2.0. Tech. rep., Technical Report NAS-95-020, NASA Ames Research Center (1995)
4. Blelloch, G.E.: NESL: a nested data parallel language. Carnegie Mellon Univ. (1992)
5. Blelloch, G.E., Anderson, D., Dhulipala, L.: Parlaylib-a toolkit for parallel algorithms on shared-memory multicore machines. In: Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures. pp. 507–509 (2020)
6. Blelloch, G.E., Sabot, G.W.: Compiling collection-oriented languages onto massively parallel computers. Journal of parallel and distributed computing **8**(2) (1990)
7. Blumofe, R.D., Leiserson, C.E.: Scheduling multithreaded computations by work stealing. Journal of the ACM (JACM) **46**(5), 720–748 (1999)
8. Chakravarty, M.M.T., Keller, G., Lee, S., McDonell, T.L., Grover, V.: Accelerating Haskell array codes with multicore GPUs. In: DAMP '11: The 6th workshop on Declarative Aspects of Multicore Programming. ACM (Jan 2011)
9. De Wolff, I.G., Keller, G.: Work assisting: Linking task-parallel work stealing with data-parallel self scheduling. In: The 10th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming. pp. 13–24 (2024)
10. De Wolff, I.G., Van Balen, D., Keller, G.: Artifact for the paper: Scheduling task and data parallelism in array languages with work assisting (Jun 2025). https://doi.org/10.5281/zenodo.15602901, https://doi.org/10.5281/zenodo.15602901
11. De Wolff, I.G., Van Balen, D.P., Keller, G.K., McDonell, T.L.: Zero-overhead parallel scans for multi-core cpus. In: Proceedings of the 15th International Workshop on Programming Models and Applications for Multicores and Manycores (2024)
12. Dotsenko, Y., Govindaraju, N.K., Sloan, P.P., Boyd, C., Manferdelli, J.: Fast scan algorithms on graphics processors. In: Proceedings of the 22nd annual international conference on Supercomputing. pp. 205–213 (2008)
13. Fluet, M., Rainey, M., Reppy, J., Shaw, A., Xiao, Y.: Manticore: A heterogeneous parallel language. In: Proceedings of the 2007 workshop on Declarative aspects of multicore programming. pp. 37–44 (2007)

14. Grelck, C., Hinckfuß, K., Scholz, S.B.: With-loop fusion for data locality and parallelism. In: Butterfield, A., Grelck, C., Huch, F. (eds.) Implementation and Application of Functional Languages. pp. 178–195. Springer Berlin Heidelberg, Berlin, Heidelberg (2006)
15. Henriksen, T., Serup, N.G., Elsman, M., Henglein, F., Oancea, C.E.: Futhark: purely functional gpu-programming with nested parallelism and in-place array updates. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 556–571 (2017)
16. Hui, R.K., Kromberg, M.J.: Apl since 1978. Proceedings of the ACM on Programming Languages **4**(HOPL), 1–108 (2020)
17. Kruskal, C.P., Weiss, A.: Allocating independent subtasks on parallel processors. IEEE Transactions on Software engineering (10), 1001–1016 (1985)
18. Loidl, H.W., Hammond, K.: On the granularity of divide-and-conquer parallelism. In: Proceedings of the 1995 Glasgow Workshop on Functional Programming (1995)
19. Matsakis, N.D., Klock, F.S.: The Rust language. ACM SIGAda Ada Letters (2014)
20. McDonell, T.L., Chakravarty, M.M., Keller, G., Lippmeier, B.: Optimising purely functional gpu programs. ACM SIGPLAN Notices **48**(9), 49–60 (2013)
21. Merrill, D., Garland, M.: Single-pass parallel prefix scan with decoupled look-back. NVIDIA, Tech. Rep. NVR-2016-002 (2016)
22. Moura, A.L.D., Ierusalimschy, R.: Revisiting coroutines. ACM Transactions on Programming Languages and Systems (TOPLAS) **31**(2), 1–31 (2009)
23. Ragan-Kelley, J., Adams, A., Sharlet, D., Barnes, C., Paris, S., Levoy, M., Amarasinghe, S., Durand, F.: Halide: Decoupling algorithms from schedules for high-performance image processing. Communications of the ACM **61**(1), 106–115 (2017)
24. Robison, A., Voss, M., Kukanov, A.: Optimization via reflection on work stealing in TBB. In: 2008 IEEE International Symposium on Parallel and Distributed Processing. pp. 1–8. IEEE (2008)
25. Sbirlea, A., Agrawal, K., Sarkar, V.: Elastic tasks: Unifying task parallelism and SPMD parallelism with an adaptive runtime. In: European Conference on Parallel Processing. pp. 491–503. Springer (2015)
26. Steuwer, M., Remmelg, T., Dubach, C.: Lift: a functional data-parallel ir for high-performance gpu code generation. In: 2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). pp. 74–85. IEEE (2017)
27. Tzannes, A., Caragea, G.C., Barua, R., Vishkin, U.: Lazy binary-splitting: a run-time adaptive work-stealing scheduler. ACM Sigplan Notices **45**(5), 179–190 (2010)
28. Van Balen, D., De Matteis, T., Grelck, C., Henriksen, T., Hsu, A.W., Keller, G.K., Koopman, T., McDonell, T.L., Oancea, C., Scholz, S.B., Sinkarovs, A., Smeding, T., Trinder, P., De Wolff, I.G., Ziogas, A.N.: Comparing parallel functional array languages: Programming and performance. arXiv preprint arXiv:2505.08906 (2025)
29. Van Balen, D., Keller, G., De Wolff, I.G., McDonell, T.L.: Fusing gathers with integer linear programming. In: Proceedings of the 1st ACM SIGPLAN International Workshop on Functional Programming for Productivity and Performance. pp. 10–23 (2024)
30. Wimmer, M., Träff, J.L.: Work-stealing for mixed-mode parallelism by deterministic team-building. In: Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures. pp. 105–116 (2011)
31. Ziogas, A.N., Schneider, T., Ben-Nun, T., Calotoiu, A., De Matteis, T., de Fine Licht, J., Lavarini, L., Hoefler, T.: Productivity, portability, performance: Data-centric python. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 1–13 (2021)